1. 上课约定须知

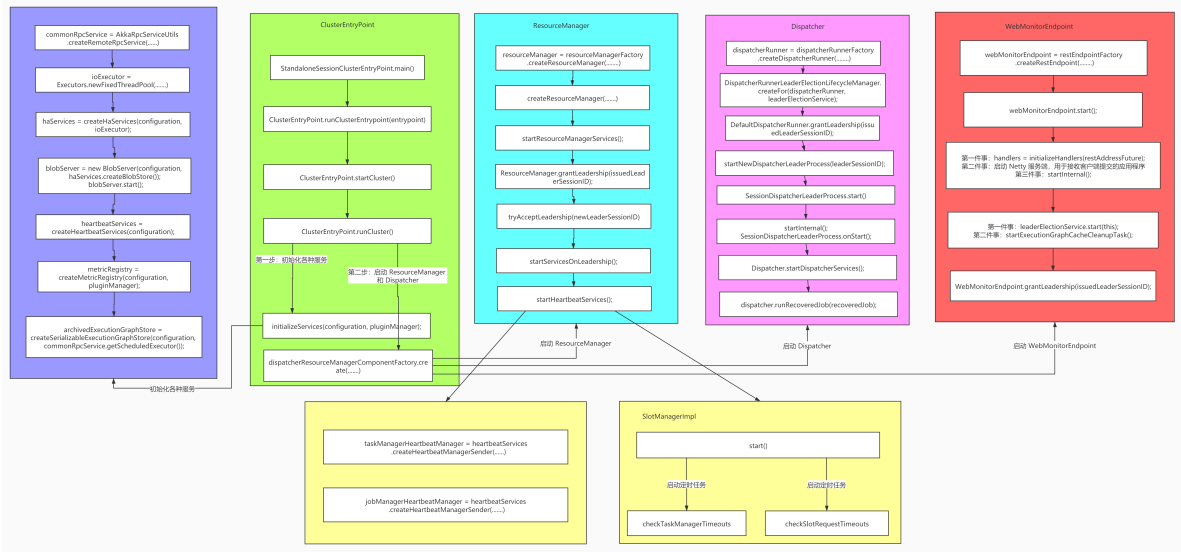
课程主题：Flink源码解析 -- 第二次课
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次内容总结

上次课程是 Flink 源码的第一次课程，主要讲解的是 Flink 源码阅读的一些基础准备：Flink 的 RPC 和集群启动脚本分析，然后重点讲解 JobManager 的启动过程的源码分析，主要内容是：

```
1、Flink RPC 详解
2、Flink 集群启动脚本分析
3、Flink 主节点 JobManager 启动分析
```

其中关于 Flink 集群的 JobManager 启动源码分析，主要包括：



3. 上次作业复盘

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

```
1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态
```

4. 本次内容大纲

今天主要的内容：Flink Standalone 集群的从节点 TaskManager 的启动

```
1、Flink TaskManager 启动源码分析
2、TaskManager/TaskExecutor 注册
3、TaskExecutor 和 ResourceManager 心跳
4、TaskExecutor 进行 Slot 汇报
```

5. 详细课堂内容

5.1. Flink TaskManager 启动源码分析

TaskManager 是 Flink 的 worker 节点，它负责 Flink 中本机 slot 资源的管理以及具体 task 的执行。

TaskManager 上的基本资源单位是 slot，一个作业的 task 最终会部署在一个 TaskManager 的 slot（物理slot，逻辑slot，slot共享） 上运行，TaskManager 会负责维护本地的 slot 资源列表，并来与 Flink Master 和 JobManager 通信。

根据 上一次 课程中启动脚本分析，得知：

```
1、主节点启动：standalonesessioin，最终的启动类：StandAloneSessionClusterEntryPorint
2、从节点启动：taskExecutor，最终启动类是：TaskManagerRunner
```

根据以上的脚本启动分析：TaskManager 的启动主类： TaskManagerRunner

在还没有看 TaskManager 从节点的源码启动分析，就能得知：

```
1、肯定要启动一些必要的服务
2、肯定要去找主节点，然后执行注册
3、当注册成功之后，就开始维持心跳
4、从节点启动好了之后还需要向主节点汇报资源的情况
```

这个情况，适用于 YARN 中的 NodeManager 的启动！

如果你肯定很清楚了 Spark Standalone 集群和 Flink Standalone 的启动过程，那么你也能很容易去了解 YARN 的启动！

```
1、主节点： ResourceManager
2、从节点： NodeManager
```

进入到 TaskMaanager 的源码分析流程：

```
TaskManagerRunner.main()
    runTaskManagerSecurely(args, ResourceID.generate());

        // 加载配置
        Configuration configuration = loadConfiguration(args);

        // 启动 TaskManager
        runTaskManagerSecurely(configuration, resourceID);

            // 启动 TaskManager
            // 这个具体实现是： 首先初始化 TaskManagerRunner， TaskManager 启动中，要干的所有的事情，都是在这个构造方法里面！
            // 最后，再调用 TaskManagerRunner.start() 来启动，但是实际上，并没有做任何有效的操作！
            runTaskManager(configuration, resourceID, pluginManager);

                // 第一步：构建 TaskManagerRunner 实例
                // 具体实现中也做了两件事：
                // 第一件事： 初始化了一个 TaskManagerServices 对象！ 其实这个动作就类似于 JobManager 启动的时候，的第一件大事！
                // 第二件是： 初始化 TaskExecutor（Standalone集群中，提供资源的角色，ResourceManager其实就是管理集群中的从节点的管理角色）
                // TaskExecutor 他是一个 RpcEndpoint，意味着，当调用完毕构造方法之后，就要去调用 onStart() 生命周期方法！
                taskManagerRunner = new TaskManagerRunner(...);

                    // 初始化一个线程池 ScheduledThreadPoolExecutor 用于处理回调
                    this.executor = Executors.newScheduledThreadPool(....)
```

```
                    // 获取高可用模式
                    highAvailabilityServices =
HighAvailabilityServicesUtils.createHighAvailabilityServices(...)

                    // 创建 RPC 服务
                    rpcService = createRpcService(configuration,
highAvailabilityServices);

                    // 创建心跳服务
                    heartbeatServices =
HeartbeatServices.fromConfiguration(conf);

                    // 创建 BlobCacheService，内部会启动两个定时任务：
PermanentBlobCleanupTask 和 TransientBlobCleanupTask
                    blobCacheService = new BlobCacheService(....)

                    // 创建 TaskManager： 实际上返回的是： TaskExecutor
                    taskManager = startTaskManager(.....)

                        // 第一件大事：初始化 TaskManagerServices
                        taskManagerServices =
TaskManagerServices.fromConfiguration(...)

                            // 初始化 TaskEventDispatcher
                            taskEventDispatcher = new TaskEventDispatcher();
                            // 初始化 IOManagerASync
                            ioManager = new IOManagerAsync(...)
                            // 初始化 NettyShuffleEnvironment
                            shuffleEnvironment = createShuffleEnvironment(...)
                            // 初始化 KVStageService
                            kvStateService =
KvStateService.fromConfiguration(...)
                            // 初始化 BroadCastVariableManager
                            broadcastVariableManager = new
BroadcastVariableManager();
                            // 初始化 TaskSlotTable
                            taskSlotTable = createTaskSlotTable(...)
                            // 初始化 DefaultJobTable
                            jobTable = DefaultJobTable.create();
                            // 初始化 JobLeaderService
                            jobLeaderService = new DefaultJobLeaderService(....)
                            // 初始化 TaskStateManager
                            taskStateManager = new
TaskExecutorLocalStateStoresManager()
                            // 初始化 LibraryCacheManager
                            libraryCacheManager = new BlobLibraryCacheManager()
                            // 返回 TaskManagerServices
                            return new TaskManagerServices(....)

                        // 第二件大事： 初始化一个 TaskExecutor
                        // TaskExecutor本身是一个 RpcEndpoint，构造方法完了之后，会调
用：onStart
                        return new TaskExecutor(.....)

                            // 初始化心跳管理器：jobManagerHeartbeatManager
                            this.jobManagerHeartbeatManager =
createJobManagerHeartbeatManager(heartbeatServices,  resourceId);
```

```
                            // 初始化心跳管理器：resourceManagerHeartbeatManager
                            this.resourceManagerHeartbeatManager =
createResourceManagerHeartbeatManager(heartbeatServices,

                        resourceId);

                            // 转到 TaskExecutor 的 onStart() 方法
                            TaskExecutor.onStart();

                                // 第一步：开启相关服务
                                startTaskExecutorServices();

                                    // 第一件事：监控 ResourceManager
                                    // 完成：注册，心跳，资源汇报
                                    resourceManagerLeaderRetriever.start(....);

                                    // 第二件事：启动 TaskSlotTable 服务
                                    taskSlotTable.start(...);

                                    // 第三件事：监控 JobMaster
                                    jobLeaderService.start(...)

                                    // 第四件事：启动 FileCache 服务
                                    fileCache = new FileCache(....)

                                // 第二步：开始注册超时检查：  5min
                                startRegistrationTimeout();

                // 第二步：启动 TaskManagerRunner
                taskManagerRunner.start();
```

TaskManagerRunner 的启动大致分为三类比较重要的：

```
1、一些基础服务
2、TaskManagerServices
3、TaskExecutor
```

Flink 集群主从架构： JobManager TaskManager

```
Flink:  ResourceManager + TaskExecutor（负责slot的管理 + 负责task的执行）
YARN:   ResourceManager + NodeManager
                ResourceManager的内部有一个：ApplicationMaster
```

不管是主节点： JobManager 还是 从节点 TaskManager , 除了关于资源的管理和调度以为，还需要一些其他的服务

Flink 的代码中，有大量的异步编程的代码。！

```
CompletableFuture, 大量提交的请求的执行，和回调的执行等等都是由线程池来搞定的!
```

```
future.xxx(() -> xxxxx(),  exceutor)
```

当一个 Flink Job 运行的时候，同样有主控程序，也有任务程序：

```
JobMaster  +  StreamTask
Driver  + Executor（Task）  MapReduce：  Task进程级别！新版本也改成了线程级别！
```

JobManager ResourceManager JobMaster

```
startJobManager() ---> new JobMaster()
```

如果你把 Job 提交给 YARN 去运行的时候： Per-Job （JobManager）类似于 Standalone 集群，ResourceManager Dispatcher 等都启动在 JobManager(JobMaster) 里面

5.2. TaskManager/TaskExecutor 注册

总结一句最重要的精髓：TaskManager 是一个逻辑抽象，代表一台服务器，这条服务器的启动，必然会包含一些服务，另外再包含一个 TaskExecutor，存在于TaskManager的内部，真实的帮助 TaskManager 完成各种核心操作：

```
1、提交Task执行
2、申请和释放slot

onRegistrationFailure(failure);
onRegistrationSuccess(result.f1);
```

核心入口为：

```
resourceManagerLeaderRetriever.start(new ResourceManagerLeaderListener());
```

记住这种代码结构：

```
1、resourceManagerLeaderRetriever 的实现类是： LeaderRetrievalService 的实现类：
ZooKeeperLeaderRetrievalService， 它是 NodeCacheListener 的子类

2、NodeCacheListener 是 curator 提供的监听器，当指定的 zookeeper znode 节点数据发生改
变，则会接收到通知回调 nodeChanged() 方法， 这个组件的作用，类似于 zookeeper 中提供的
watcher 组件

3、在 nodeChanged() 方法中会调用对应的 LeaderRetrievalListener 的
notifyLeaderAddress() 方法

4、ResourceManagerLeaderListener 是 LeaderRetrievalListener 的子类

5、resourceManagerLeaderRetriever 进行监听，当发生变更的时候，就会回调：
ResourceManagerLeaderListener 的 notifyLeaderAddress 方法
```

这在注册监听之后，会收到一个响应，然后回调：

```
ResourceManagerLeaderListener.notifyLeaderAddress();
```

内部详细实现：

```
// 关闭原有链接
closeResourceManagerConnection(cause);

// 开启注册超时任务
startRegistrationTimeout();

// 当前 TaskExecutor 完成和 ResourceManager 的链接
tryConnectToResourceManager();
```

最重要的是第三步，TaskExecutor 和 ResourceManager 建立连接，会进行注册，心跳，Slot汇报 三件大事!

进入到:

```
TaskExecutor.connectToResourceManager();
```

内部实现:

```
// 生成 TaskExecutor 注册对象
final TaskExecutorRegistration taskExecutorRegistration = new
TaskExecutorRegistration(
    getAddress(),
    getResourceID(),
    unresolvedTaskManagerLocation.getDataPort(),
    hardwareDescription,
    taskManagerConfiguration.getDefaultSlotResourceProfile(),
    taskManagerConfiguration.getTotalResourceProfile()
);

// 生成链接 TaskExecutorToResourceManagerConnection
resourceManagerConnection = new TaskExecutorToResourceManagerConnection(
    log, getRpcService(),
    taskManagerConfiguration.getRetryingRegistrationConfiguration(),
    resourceManagerAddress.getAddress(),
    resourceManagerAddress.getResourceManagerId(),
    getMainThreadExecutor(),
    new ResourceManagerRegistrationListener(),
    taskExecutorRegistration
);

// 进行连接，并注册
resourceManagerConnection.start();
```

进入到具体实现:

```
// 生成注册行动对象
final RetryingRegistration<F, G, S> newRegistration = createNewRegistration();
    // 内部调用： generateRegistration() 方法帮忙生成： newRegistration
    RetryingRegistration<F, G, S> newRegistration =
checkNotNull(generateRegistration());
    // 注意后面的回调：这个回调会在注册成功之后，帮忙完成 slot 汇报
    onRegistrationSuccess(result.f1);

// 开始注册
newRegistration.startRegistration();
```

```
    // 内部实现：  先建立和 ResourceManager 的 RPC 链接
    rpcService.connect(....);
    // 进行注册
    register(rpcGateway, .....);
        // 内部实现
        registrationFuture = invokeRegistration(gateway, fencingToken,
timeoutMillis);
            // 给 ResourceManager 发送 RPC 注册请求
            resourceManager.registerTaskExecutor(taskExecutorRegistration,
timeout);
                // ResourceManager 执行注册处理

ResourceManager.registerTaskExecutorInternal(taskExecutorGateway,
taskExecutorRegistration);
                    // 完成注册
                    WorkerRegistration<WorkerType> registration = new
WorkerRegistration<>(
                        taskExecutorGateway, newWorker,
taskExecutorRegistration.getDataPort(),
                        taskExecutorRegistration.getHardwareDescription()
                    );
                    taskExecutors.put(taskExecutorResourceId, registration);
                    // 维持心跳

taskManagerHeartbeatManager.monitorTarget(taskExecutorResourceId, new
HeartbeatTarget<Void>() {
                        @Override
                        public void receiveHeartbeat(ResourceID resourceID, Void
payload) {
                        }
                        @Override
                        public void requestHeartbeat(ResourceID resourceID, Void
payload) {
                            // 给 TaskExecutor 发送心跳请求

 taskExecutorGateway.heartbeatFromResourceManager(resourceID);
                        }
                    });
                    // 返回注册成功的消息
                    return new
TaskExecutorRegistrationSuccess(registration.getInstanceID(), resourceId,
clusterInformation);
```

5.3. TaskExecutor 和 ResourceManager 心跳

5.3.1. ResourceManager 端心跳服务启动

ResourceManager 在初始化的最后，执行了：

```
ResourceManager.startHeartbeatServices();
```

启动了两个心跳服务：

```
// 维持 TaskExecutor 和 ResourceManager 之间的心跳
taskManagerHeartbeatManager =
heartbeatServices.createHeartbeatManagerSender(resourceId, new
TaskManagerHeartbeatListener(), getMainThreadExecutor(), log);

// 维持 JobMaster 和 ResourceManager 之间的心跳
jobManagerHeartbeatManager =
heartbeatServices.createHeartbeatManagerSender(resourceId, new
JobManagerHeartbeatListener(), getMainThreadExecutor(), log);
```

具体是构造了一个 HeartbeatManagerSenderImpl 实例对象，并且调用了：

```
mainThreadExecutor.schedule(this, 0L, TimeUnit.MILLISECONDS);
```

这句代码的意思表示，立即调用：HeartbeatManagerSenderImpl .run() 执行一次！那么请看：run() 方法的内部具体实现：

```
// 给每个心跳目标对象发送心跳请求
for(HeartbeatMonitor<O> heartbeatMonitor : getHeartbeatTargets().values()) {
    requestHeartbeat(heartbeatMonitor);
}

// 延迟 heartbeatPeriod 之后，再次执行 run() 方法，实现了循环
getMainThreadExecutor().schedule(this, heartbeatPeriod, TimeUnit.MILLISECONDS);
```

在此，需要特别注意：heartbeatMonitor 就是一个 heartbeatTarget，每个 heartbeatTarget 其实就是一个 TaskExecutor，这个可以在 ResourceManager 端完成 TaskExecutor 注册的时候进行验证。

当 ResourceManager 端完成一个 TaskExecutor 的注册的时候，马山调用：

```
// 维持心跳
taskManagerHeartbeatManager.monitorTarget(taskExecutorResourceId, new
HeartbeatTarget<Void>() {
    @Override
    public void receiveHeartbeat(ResourceID resourceID, Void payload) {
    }
    @Override
    public void requestHeartbeat(ResourceID resourceID, Void payload) {
        // 给 TaskExecutor 发送心跳请求
        taskExecutorGateway.heartbeatFromResourceManager(resourceID);
    }
});
```

这样子，刚才注册的 TaskExecutor 就先被封装成一个 HeartbeatTarget， 然后被加入到 taskManagerHeartbeatManager 进行管理的时候，变成了 HeartbeatMonitor。当这句代码完成执行的时候，当前 ResourceManager 的心跳目标对象，就多了一个 TaskExecutor，然后当执行：

```
taskExecutorGateway.heartbeatFromResourceManager(resourceID);
```

就给 TaskExecutor 发送了一个心跳请求。

5.3.2. TaskExecutor 端心跳

当 TaskExecutor 接收到 ResourceManager 的心跳请求之后，进入内部实现：

```
TaskExecutor.heartbeatFromResourceManager(ResourceID resourceID);
    // 内部实现
    resourceManagerHeartbeatManager.requestHeartbeat(resourceID, null);
        // 内部实现
        reportHeartbeat(requestOrigin);
            // 内部实现
            heartbeatMonitor.reportHeartbeat();
                // 内部实现
                lastHeartbeat = System.currentTimeMillis();
                // 重设心跳超时相关的 时间 和 延迟调度任务
                resetHeartbeatTimeout(heartbeatTimeoutIntervalMs);
```

如果连续 5 次心跳请求没有收到，也就是说，如果 50s 内都没有收到心跳请求，则执行心跳超时处理。

```
heartbeatListener.notifyHeartbeatTimeout(resourceID);
```

超时处理也非常的暴力有效，Flink 认为： 如果 TaskExecutor 收不到 ResourceManager 的心跳请求了，则认为当前 ResourceManager 死掉了。但是 Flink 集群肯定会有一个 active 的 ResourceManager 节点的。而且之前也注册过监听，所以一定已经收到过通知了，然后现在需要做的，只是重新链接到 新的 active ResourceManager 即可：

```
reconnectToResourceManager(new TaskManagerException(String.format("The heartbeat
of ResourceManager with id %s timed out.", resourceId)));
```

Flink 心跳机制，和 HDFS 的心跳机制不一样。

```
1、HDFS 的心跳是：namenode 率先启动，然后启动一个超时检查服务，然后 datanode 启动之后过来注册，当注册成功之后，datanode就是执行定时心跳任务，这种模式中，是 从节点 datanode主动！

2、Flink 的心跳是：Resourcemanager 率先启动，然后启动一个 向所有心跳目标对象发送心跳请求 的定时任务。当有 TaskExecutor 上线并注册成功，则会生成一个 HeartBeatMonitor 加入到心跳目标对象集合，然后 Resourcemanager 开始一视同仁的向所有 TaskExecutor 发送心跳请求。
TaskExecutor 接收到心跳请求，则执行最近心跳时间的修改，和心跳超时定任务的重置。如果超时了，则发起请求，链接新的 resourcemanager。
```

5.4. TaskExecutor 进行 Slot 汇报

当注册成功，ResourceManager 会返回 TaskExecutorRegistrationSuccess 对象！

然后回调下面的方法，进入到 slot 汇报的过程。

```
TaskExecutorToResourceManagerConnection.onRegistrationSuccess(TaskExecutorRegist
rationSuccess success);
    // 继续回调
    ResourceManagerRegistrationListener.onRegistrationSuccess(this, success);
        // 封装链接对象
        establishResourceManagerConnection(resourceManagerGateway,
resourceManagerId, taskExecutorRegistrationId, ....);
            // 内部实现
            resourceManagerGateway.sendSlotReport(
                getResourceID(),
                taskExecutorRegistrationId,
                taskSlotTable.createSlotReport(getResourceID()),
taskManagerConfiguration.getTimeout()
            );
```

在上面的代码中，先调用：taskSlotTable.createSlotReport(....) 生成 SlotReport 对象！然后通过 sendSlotReport() RPC 请求发送给 ResourceManager。

具体实现：

```
if(slotManager.registerTaskManager(workerTypeWorkerRegistration, slotReport)) {
    onTaskManagerRegistration(workerTypeWorkerRegistration);
}
```

内部具体实现：

```
for(SlotStatus slotStatus : initialSlotReport) {
    registerSlot(slotStatus.getSlotID(), slotStatus.getAllocationID(),
slotStatus.getJobID(), ...., taskExecutorConnection);
        createAndRegisterTaskManagerSlot(slotId, resourceProfile,
taskManagerConnection);
            final TaskManagerSlot slot = new TaskManagerSlot(slotId,
resourceProfile, taskManagerConnection);
            slots.put(slotId, slot);
}
```

到此为止，slot 的汇报就完成了！

6. 本次课程总结

本次课程的主要内容：TaskManager 的启动，主要内容有：

```
1、Flink TaskManager 启动源码分析
2、TaskManager/TaskExecutor 注册
3、TaskExecutor 和 ResourceManager 心跳
4、TaskExecutor 进行 Slot 汇报
```

到此为止，把 Flink 的 Standalone 集群启动的流程基本讲完。最终，总结，其实不管是主节点 JobManager 还是 TaskManager 在启动过程中，都是提前启动一些服务来为将来的 Job 提交执行做准备。

大致总结一下：

JobManager 的核心作用：

```
1、启动了 WebMonitorEndppint 来接收客户端的 rest 请求
2、启动了 ResourceManager 来管理集群的所有资源，资源使用 slot 的抽象来进行管理
3、启动了 Dispatcher 来负责调度 Job 执行
```

TaskManager 的核心作用：

```
1、启动 TaskManagerServices，其实是启动很多服务组件
2、启动 TaskExecutor，进行资源抽象封装和注册，并维持心跳
```

其实 TaskManager 还有其他重要的作用，比如：

```
1、Slot 资源管理（申请和释放）
2、Task 提交和执行
3、State 和 Checkpoint 相关动作
```

等这些工作机制的源码分析，会在 Job 提交和执行的时候，再讲！

7. 本次课程作业

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

```
1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态
```